Since you are new to game design, you decide to skip creating a player controller and dive right into creating your dungeons. Sadly, you’re stuck at home right now, and can’t have your friends playtest your dungeons. So, you decide to create a program that will judge whether or not your dungeons are good. This may not be the best idea, but you do what you can.

You’re using an undirected graph to represent your dungeons, with vertices representing each room of the dungeon, and edges representing the connecting halls. From your vast amount of dungeon experience, you boil a good dungeon down to one thing: a non-linear design. This means that a good dungeon will have paths that lead back to rooms you’ve already been to.

All the dungeons in the game are contained in the Game class, which holds a graph. Each dungeon is a subgraph of the Game graph that isn't connected to other subgraphs. Thus, a Game graph may contain several dungeons, each being a subgraph disconnected from other dungeons.

Your program will need to process all of the dungeons (subgraphs) in the Game graph. With CSE 331 being one of your favorite classes ever, you know that Professor Onsay has provided all the information you’ll need to solve this. Should be a piece of cake!

# **Challenge**

## **Overview**

In this coding challenge, you will be creating a function that can (i) traverse a graph made up of smaller subgraphs (dungeons) and (ii) determine if each subgraph (dungeon) has paths that loop back to an already visited vertex. If a given subgraph (dungeon) has a path that loops back to an already visited vertex, that will be considered a good subgraph (dungeon). The function should return a count of how many good subgraphs (dungeons) the Game graph as a whole contains.

You will be implementing the **count\_good\_dungeons** function that takes in a variable, **game**, containing a graph object holding all of the subgraphs (dungeons) in the game that you will be crawling. This function will return an integer that is equal to the count of how many of its subgraphs (dungeons) contain paths that loop back on themselves.

You are provided a premade Game class that is implemented using an adjacency list. The Game graph is made up of room nodes and that class is also provided. These classes and their functions can be found in the game.py file.

*Modify the following function*

**count\_good\_dungeons(game: Graph) -> int**

* **game: Graph:** A graph that stores the subgraphs (dungeons) to check. Note that this graph may contain multiple separate subgraphs (dungeons).
* **Return:** An integer that represents the count of how many good dungeons were found
* **Time Complexity:** **O(V + H))** where V is the number of vertices in the graph and H is the number of hallways or edges in the graph
* **Space Complexity:** **O(V)** where V is the number of vertices in the graph

*Do* ***NOT*** *modify the following functions. You will, however, need to use them in your solution.*

**class Room:**

This class represents a room in a dungeon. It functions as a node/vertex in a graph and contains a **room\_id** and a list of the rooms adjacent to it. It contains a function that will add adjacent rooms.

* **Attributes:**
  + room\_id: Any Hashable Type: The identifying value of the room typically will be a string or an int, but could be anything. As long as it’s hashable
  + adjacent\_rooms: List[Room]: A list of adjacent room objects
* **\_\_init\_\_(self, name: Any Hashable Type) -> None:**
  + Constructs a room object with the given name as the room\_id
  + **room\_id: Any Hashable Type:** The identifying value of the room typically will be a string or an int, but could be anything. As long as it’s hashable
  + **adjacent\_rooms: List[Room]:** A list of adjacent room objects
  + **\_adj\_rooms\_set: Set[ room\_id]:** A set of adjacent room ideas. This is a private variable and should NOT be accessed outside of this class.
  + **Return:** None
* **add\_adjacent(self, adj\_node: Room) -> bool:**
  + If the given node is not already in the room’s adjacent rooms list then it will be added and True will be returned. Otherwise, it will not be added and False will be returned
  + **adj\_node: Room:** The room object to add to the room’s adjacent list
  + **Return:** A boolean representing whether or not the room could be added to adjacent list

**class Game:**

This class represents a graph implemented using an adjacency list and contains all of the rooms in the game. Rooms connect with one another to form dungeons. Note that you may not be able to travel from one node across the graph to another because each dungeon is a “subgraph” in the overall game graph. This class contains the functions **add\_to\_game** and **add\_hallway**.

* **Attributes:**
  + rooms: List[Room]: A list of all the room objects in the game.
* **\_\_init\_\_(self) -> None:**
  + This function will initialize a Game object and take in nothing as input
  + **rooms: List[Room]:** A list of all the room objects in the game.
  + **\_rooms\_set: Set[room\_id]:** A set containing the id’s of all the rooms in the game. This is a private variable and should NOT be accessed outside of the Game class.
  + **Return:** None
* **add\_to\_game(self, room\_id) -> bool:**
  + Creates and adds a new room to the game with the provided room\_id. Will return True if the room\_id already exists in the game, False otherwise.
  + **room\_id: Any Hashable Type:** The value that will be used to create and identify the the new room
  + **Return**: A bool representing whether or not the room could be added to the game’s room list.
* **add\_hallway(self, start: room\_id, end: room\_id) -> bool:**
  + This function will add a hallway between the rooms with the given room\_ids. If the hallway between the rooms could be created True will be returned, otherwise False will be returned.
  + **room1: room\_id:** The room\_id of the first node to create the hallway to.
  + **room2: room\_id:** The room\_id of the second node to create the hallway to.
  + **Return:** A bool representing whether or not a hallway could be added connecting room1 and room2.

#### **Guarantees**

* No two rooms will have the same room\_id; all room\_ids will be **unique**
* A room will never have a hallway directly to itself. For example: Room(A) can **not** be adjacent to Room(A)

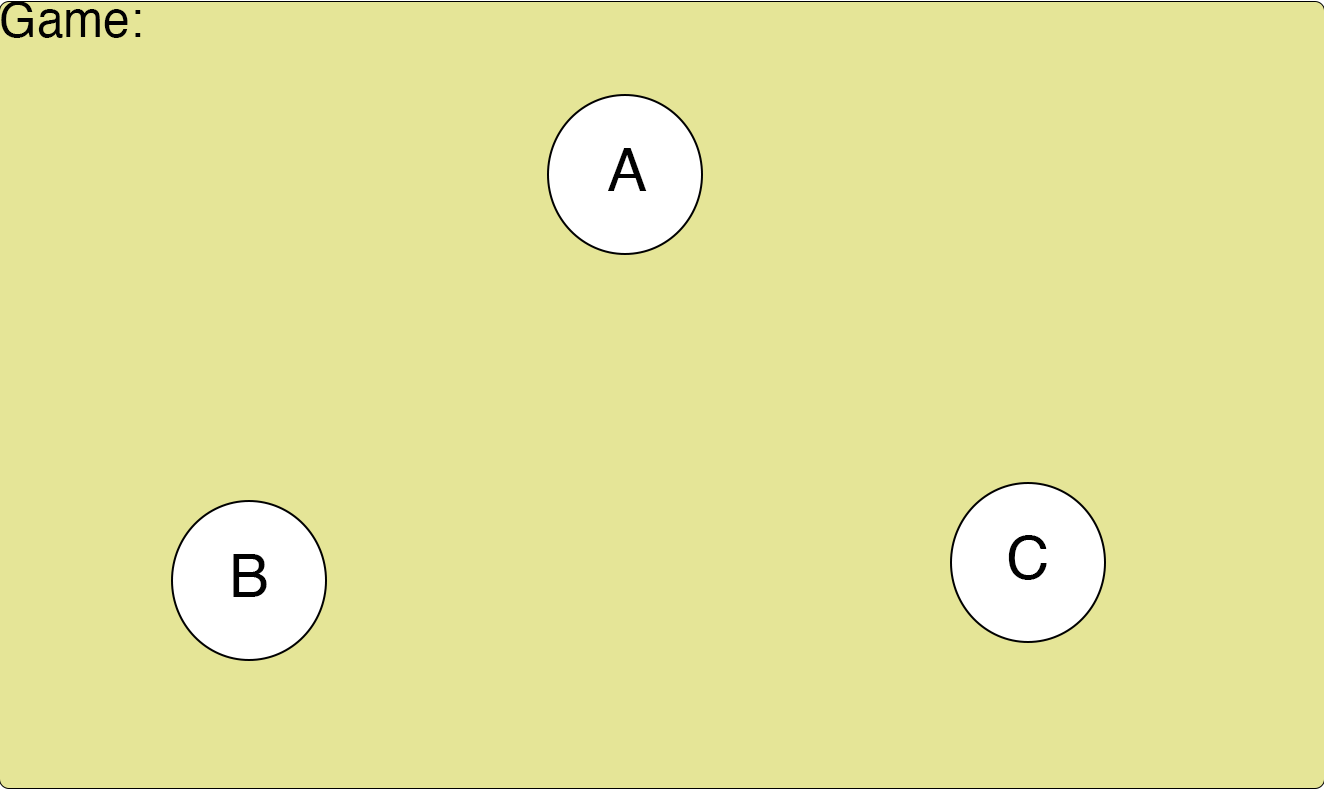
#### **Examples:**

**Ex1:** Empty. No Dungeons



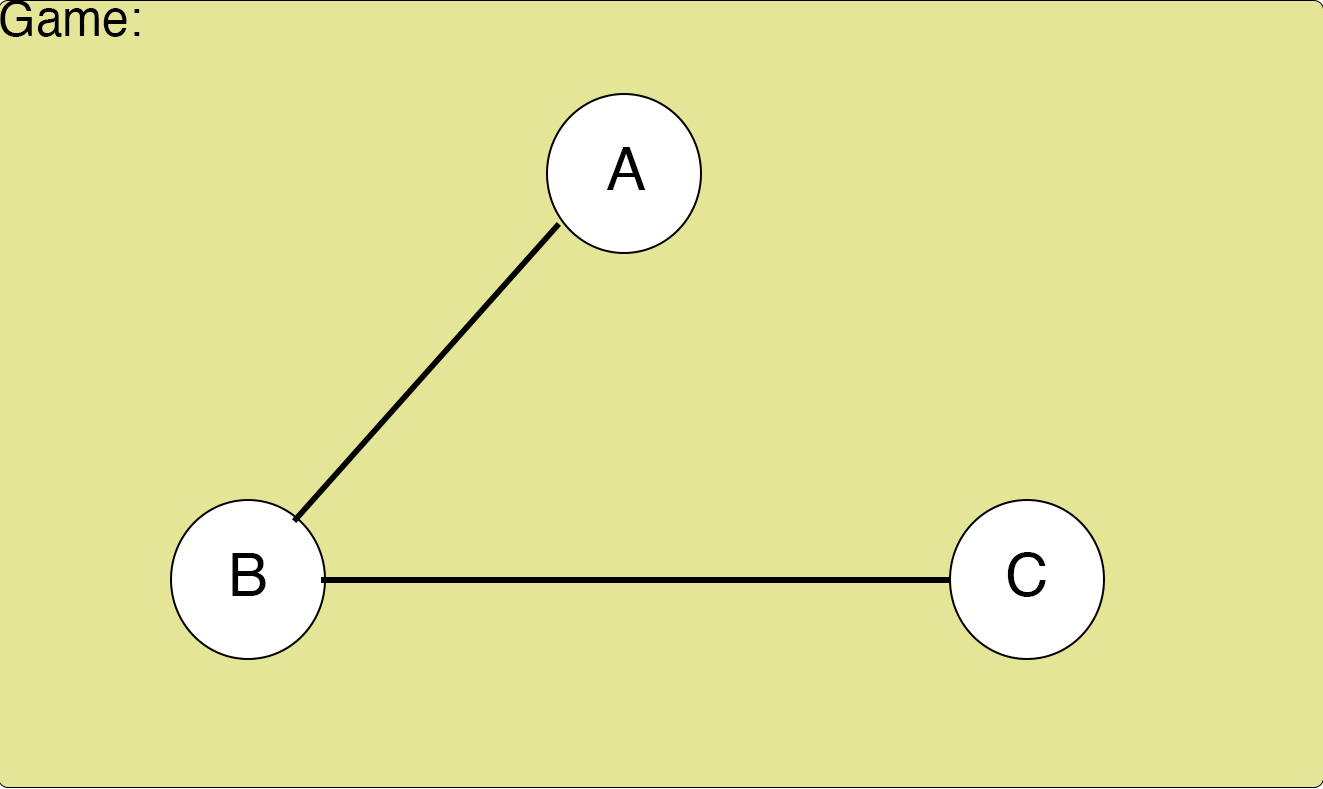
There are no rooms in this game, so the returned good dungeon count should be 0.

**Ex 2**: Three Rooms Not Connected. Three Dungeons.



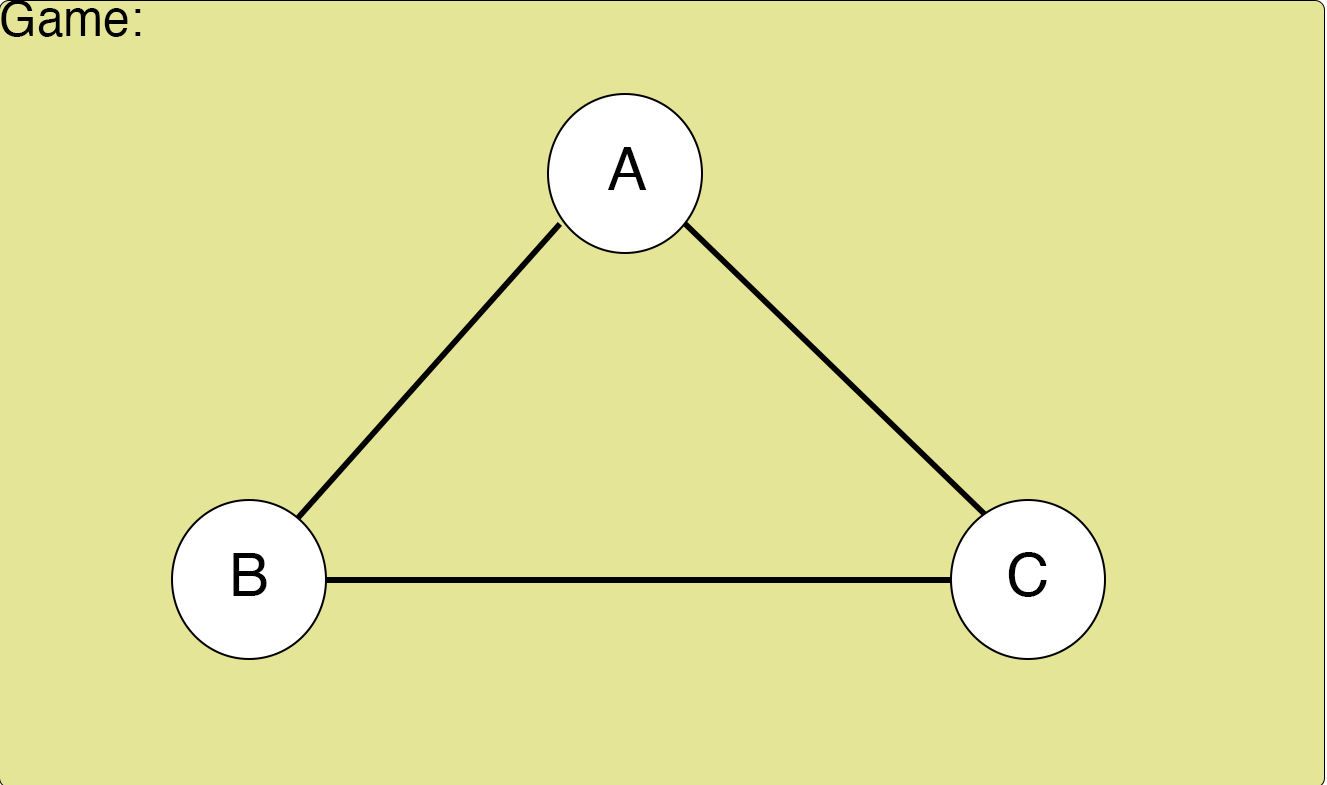
Three rooms exist in the game; however they are not connected so they comprise three separate dungeons. Since there are no hallways to create a path that leads back to a node, the good dungeon count is 0.

**Ex 3:** Three Rooms. Rooms A and B are connected and Rooms B and C are connected. One Dungeon.



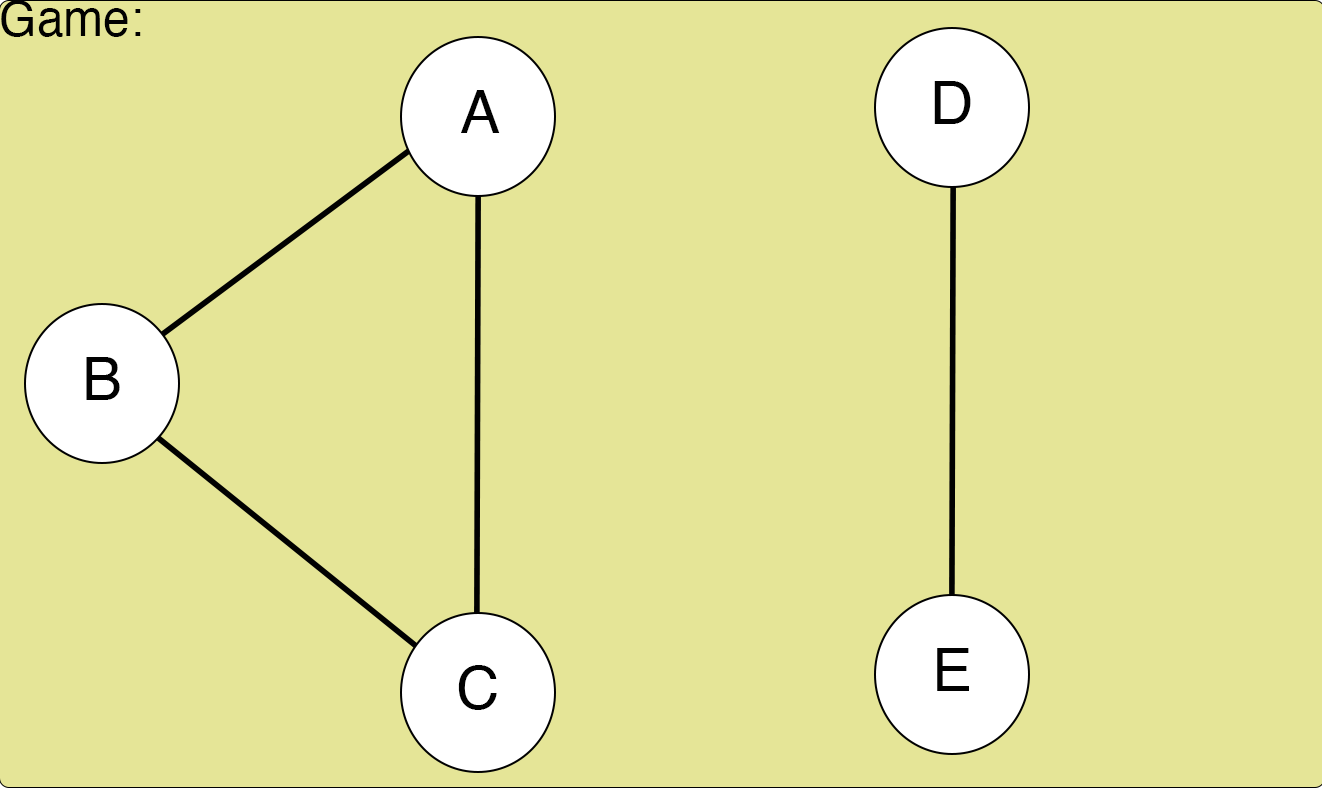
Three rooms exist in the game, and since they are all connected in some way, one dungeon exists. However, a Path going from one node to another does not create a path that loops back. So, the returned good dungeon count is 0.

**Ex 4:** Three Rooms. A is connected to B, B is connected to C, and C is connected to A. One Dungeon.



Three rooms exist in the game, and since they are all connected in some way, one dungeon exists. A path that will come back to a room that already exists in the path can be found, so the good dungeon count is 1.

**Ex 5:** Five Rooms. A is connected to B, B is connected to C, and C is connected to A. D is connected to E. 2 Dungeons



Five rooms exist in the game. Rooms A, B, and C connect to form a dungeon, and nodes D and E are connected to form a dungeon. However, dungeon ABC doesn’t connect to dungeon DE, so two dungeons exist in this game. In dungeon ABC, a path that will come back to a room that already exists in the path can be found, so the good dungeon count is incremented to 1. However, in dungeon DE, the only paths are E->D and D->E, so no path comes back to a dungeon already in the path. The good dungeon counter stays at 1, and 1 is returned.